Functional Metaprogramming in C++ and Cross-Lingual Development with HASKELL

Seyed Hossein HAERI (Hossein)[1] and Sibylle Schupp[2]

[1] MuSemantik Ltd, 6.04 Appleton Tower, 11, Crichton Street, Edinburgh, UK
hossein@musemantik.com

[2] Institute for Software Systems, School of Electrical Engineering and Information Technology, Hamburg University of Technology, Germany
schupp@tu-harburg.de

Abstract. Template and Preprocessor Metaprogramming are both well-known in the C++ community to have much in common with Functional Programming (FP). Recently, very few research threads on underpinning these commonalities have emerged to empower cross-development of C++ Metaprogramming (C++MP) and FP. Yet, self-contained real-world demonstrations on the commonalities which can truly exemplify the Functional nature of C++MP are very rare. In this paper, we provide such a demonstration with a running application which we program side-by-side: We develop a compile-time abstract datatype for Rational Numbers in C++. We then present the equivalent HASKELL runtime program to outline the FP techniques used. Expectably, our HASKELL program is considerably more succinct than its C++ counterpart metaprogram. Earlier studies spotting this relative brevity suggest methods for automated transformation of HASKELL code into C++MP. This paper studies the impediments to suggest semi-automatic translation with a multiparadigm FP language like F#.

1 Introduction

In 1994, Unruh wrote a C++ program which was designed to emit some prime numbers as warning messages [20]. These prime numbers were calculated at compile-time using techniques which are well-known in today's template metaprogramming. A year later, Veldhuizen introduced expression templates to the world of C++ metaprogramming [22]. Austern's book [3] exemplified some commonalities between STL (the Generic Programming part of the 1998 C++ Standard [1] library) and FP. Alexandrescu's book presented a *tour de force* of C++MP and was the first to explain some similarities between that and FP. Fast growth of the Boost C++ libraries catered a handful of metaprogramming libraries so well that Abrahams and Gurtovoy devoted their book [2] to that.

Recently, a new trend of FP-*injection* has started to emerge in the C++ community. All that is based on the common belief that C++MP is essentially FP but at the metaprogramming level. Yet, the limited references in support of that belief fit into two groups, neither of which suits a new C++ programmer wanting to delve into the topic: Ones which hardly scratch the surface by providing examples on say how to implement simple compile-time functions such as factorial. Or, sizeable manuals of the relevant

Boost libraries such as MPL[5], Fusion[6], Proto[11], and Preprocessor[7]. Functional programmers approaching C++ to examine that belief face the same difficulty.

What is missing is a self-contained real-world explanation of the FP techniques used in C++MP which is short and approachable on the one hand, and inclusive on the other. In this paper, we aim to fill that gap by showing the important bits of a compile-time abstract datatype representing Rational Numbers ($\mathbb{Q}$) in C++. [3] Support for $\mathbb{Q}$ is so important that `Rational` is already a built-in type in HASKELL. Furthermore, in C++, Boost.Rational[12] is a relatively old hand member of the Boost library. More to the point is that Boost.Ratio[23] has also recently been added to the Boost library to support Rational arithmetics at compile-time.

We do not aim to provide the most efficient or most facilitating implementation. In fact, both `Rational` of HASKELL and Boost.Ratio outperform our solution from several standpoints. None of the techniques we present are new either. We leverage our compile-time presentation for $\mathbb{Q}$ to demonstrate the FP techniques commonly used in C++MP. So, we go as deep into the technical details as required by a minimalist comparison.

On the other hand, with FP becoming more popular in the C++ community, attempts on automatic translation from HASKELL to C++ are also gaining gravity. For reasons that we explain in the conclusions, we believe that a bidirectional translation across HASKELL and C++ is more productive. Every now and then, we explore the subtle differences between the two implementations to discuss the impediments the respective part may cause to automatic translation in either direction. Our list of explained impediments is summarised in the conclusions.

A note on laziness is relevant here: C++ templates are lazy from many viewpoints. For example, most of the type-checking is delayed until instantiation. This is the basis of how Sipos et al. [19] managed to implement compile-time infinite sequences in C++. However, as also alluded to by Sinkovics [16], although C++MP is a lazy FP language with selective strictness, eager template evaluation is often enforced predominantly. Namely, in practice, one can consider C++MP a strict FP language with selective laziness. In the conclusion, we argue thus that translation between C++ metaprograms and **F#** programs works much better. Even though, we develop this paper on comparison between HASKELL and C++ for two reasons: Firstly, HASKELL is a pure FP language and therefore better for showing correspondence with FP. Secondly, we aim to demonstrate the inadequacy of a uniparadigm FP language for facilitating C++MP via a thorough comparison with such a language – HASKELL in this case.

In our implementation, we call our HASKELL algebraic datatype `Fraction` because `Rational` is already a built-in HASKELL type. On the contrary, we call our C++ abstract datatype `Rational` to give less grounds for name confusion. In this paper, we do not use the *0x* features of C++. In particular, as opposed to the C++0x *parameter pack* token, ellipsis here is our informal representation of unimportant details. We will drop namespace qualifiers (i.e., `std::` and `boost::`) throughout our C++ codes for brevity reasons. Due to space constraints, in our C++ codes, we use multiple common conventions for breaking long statements into multiple lines. For the same reason, when the readability is not particularly impinged, we make the code extra compact in common ways.

[3] For simplicity, in this paper, we do not distinguish between the Set-Theoretical notion of Rational Numbers and our codes representing that for programming purposes.

This paper starts by reviewing the related work in Section 2. Then, we briefly introduce the relevant parts of C++ and FP in Section 3. A short account of the parts of Boost.MPL that we use is also given in this section. The most important part of this paper is Section 4. Here, we explain how to represent $\mathbb{Q}$ as an abstract datatype both at the metaprogramming level in C++ and the ordinary level in HASKELL. We compare the implementations to enumerate the impediments to automatic translation across the languages. A detailed discussion is provided in the Conclusion.

2 Related Work

Sipos et al. [19] start by enumerating some similarities between template metaprogramming and FP. They then informally describe a method for systematically producing metafunctions out of functions written in the pure FP language CLEAN [13]. They advertise that their `Eval<>` metafunction evaluates the produced metaprograms according to the operational semantics of CLEAN, which is provided by van Eekelen and de Mol [21]. Whilst they do not formally present their operational semantics, their informal explanation of that suggests remarkable differences between the operational semantics of CLEAN and that of theirs.

$$\frac{\{f,x\} \subseteq dom(\Gamma)}{\Gamma : f\ x \Downarrow \Gamma : f\#x}\ (\textbf{bind}) \qquad \frac{\Gamma : f\ x \Downarrow \Delta : v}{\Gamma : f\#x \Downarrow \Delta : v}\ (\beta_{@}) \qquad \frac{\Gamma : f \lightning x \Downarrow \Delta : v}{\Gamma : f\#x \Downarrow \Delta : v}\ (\beta_{\lightning})$$

$$\frac{\Gamma : f \Downarrow \Theta : \lambda y.e \quad \Theta : e[x/y] \Downarrow \Delta : v}{\Gamma : f\ x \Downarrow \Delta : v}\ (@)$$

$$\frac{\Gamma : f \Downarrow \Theta_1 : \lambda y.e \quad \Gamma : x \Downarrow \Theta_2 : v_x \quad \Theta_1 \bowtie \Theta_2 : e[v_x/y] \Downarrow \Delta : v}{\Gamma : f\ x \Downarrow \Delta : v}\ (\lightning)$$

Fig. 1. Function Application of Sipos et al.

CLEAN inherits the (**app**) of Launchbury for lazy evaluation [8]. Function application in the suggested operational semantics of Sipos et al. takes several forms as depicted in Figure 1. Their $(@)$ is essentially the (**app**) rule of Launchbury. They also add $(\lightning)$ for strict application. This mere addition makes equivalence of their operational semantics and that of CLEAN questionable. It is noteworthy that the operational semantics of van Eekelen and de Mol [21] suffers from increase of expressiveness upon evaluation of let-expressions [15]. Counter-intuitively enough, in such an operational semantics, $e \lightning x$ is not proven to be observationally equivalent to x seq $e\ x$. Finally, their (**bind**) rule is simply to optionally postpone function application.

Sinkovics [16] offers certain solutions for improving the FP support in MPL and discusses why they are needed. Sinkovics and Porkoláb [18] advertise implementation of a λ-expression library on top of the operational semantics of Sipos et al. for embedded FP in C++. They also later advertise [24] extension of their λ-expression library to full support for HASKELL. None of their works are unfortunately available online for experimentation. We can therefore not evaluate their works any further. [4] Sinkovics [17] offers a restricted solution for emulating `let`-bindings in template metaprogramming.

Milewski [9] has a number of posts on his personal blog that speak about Monads [10], their benefits for C++, and how to implement Monadic entities in C++. He also has a post on how template metaprogramming with the newly added C++ construct *variadic templates* is similar to lazy list processing in HASKELL. Finally, Sankel [14] shows how to implement algebraic datatypes in C++.

3 Preliminaries

Templates were originally to enable compile-time *genericity by type* [4, §2.2] in C++, which, in crude terms, is compile-time code reuse for every type. The `struct S` (line 1 below), for example, is meant to work for every type `T1` and `T2`. Likewise, function `f` (line 2 below) is meant to work for every type `T`:

```
1 template <typename T1, typename T2> struct S {typedef ... type;};
2 template <typename T> void f(T) {...}
3 template<typename T> struct S<T, int> {...};
4 template<> void f(float) {...}
5 template<int n> class C {typedef typename S<...>::type type};
```

Templates can be specialised for types implementations of which may differ from the general one. (See the above lines 3 and 4 for the syntax.) Pattern matching is used to choose between the available implementations. Yet, C++ always chooses the best-match in pattern matching, whilst in many FP languages including HASKELL, the earliest match is always chosen. Templatisation can be over compile-time integral constants too (such as class `C` above) for which specialisation is also allowed.

Templates can have nested types/values. For example, the structure `S` above defines `type` as a nested type. When a compiler fails to infer whether a token is a nested type or other sorts of nested entity, use of the keyword `typename` informs the compiler that a nested type is the intention. (See line 5 above for example.)

MPL [5] is the metaprogramming component of the Boost C++ library. We use the following items from MPL, which we explain very briefly:

- `integral_c<T, n>` is a type representation for a compile-time constant `n` of integral type `T`.
- `bool_<b>` is a type representation for the compile-time Boolean constant `b`. `bool_<>` uses `true_` and `false_` as its compile-time equivalents for `true` and `false`.
- `not_<T>::type` is a type representing the Boolean complement of the one represented by `T`.
- `if_c<b, T1, T2>::type` is equivalent to `T1` if `b` is true, and to `T2` otherwise.

An FP language which does not allow side-effects is called a *pure* FP language. When an argument is only evaluated if needed and the result of evaluation is shared thereafter in the scope, the variable is said to be evaluated *lazily* [8].

[4] Through personal email exchange, we were informed that they are elaborating on their developments. The result is to be placed online for experimentation.

4 Compile-Time Rational Numbers

This section starts by discussing how to implement the auxiliary functions needed. It then explains how to represent $\mathbb{Q}$ itself in C++ and HASKELL. Finally, it explores the implementation techniques we use for supporting operations on $\mathbb{Q}$ in either language.

Throughout this section, we run a comparative study of the implementation techniques used across the two languages. With this study of commonalities and differences, we discuss the impediments to automatic translation between C++MP and HASKELL.

4.1 Greatest Common Divisor

Fraction cancellation is the cornerstone of any arithmetic on $\mathbb{Q}$ – that is, dividing the nominator and denominator by their greatest common divisor (*gcd*). The metafunction implementation of *gcd* using Euclid's famous algorithm is a straightforward pattern matching – the first FP technique used in C++MP that we present:

```
template<long unsigned a, long unsigned b>
struct GCD {const static long unsigned value = GCD<b, a % b>::value;};
template<long unsigned a>
struct GCD<a, 0> {const static long unsigned value = a;};
```

This is one of the very few examples where the mathematical definition of an operation is not far from its C++ metafunction implementation. The HASKELL implementation is even better and is literally *identical* to Euclid's algorithm. Both implementations use tail recursion, obviously on immutable data:

```
gcd a 0 = a
gcd a b = gcd b (a 'mod' b)
```

An important difference between C++ and HASKELL here is that, in C++, the general definition of a template needs to be introduced first. Only then can the specialisations come but their relevant order of definition is not significant so long as they are in scope. In HASKELL, however, the order of pattern matches is significant and that does include the general case. Here, for instance, the general case needs to come after the case for $gcd(a,0)$, or the recursion will never end.

We use the following example to demonstrate the significance of this difference in action when considering automatic translation. Let us suppose – for the sake of argument – that $gcd(0,a) = 0$. In order to accommodate that, one would need to also provide the following two specialisations for the C++ metafunction:

```
template<long unsigned a>
struct GCD<0, a>{const static long unsigned value = 0;};

template<> struct GCD<0, 0> {const static long unsigned value = 0;};
```

and the order of the template specialisations is not relevant because all the in-scope specialisations will be equally visible at the instantiation time. On the other hand, for the HASKELL counterpart, the following two lines would have needed to be added **before** the general case, for HASKELL always goes for the first pattern match:

```
gcd 0 a = 0
gcd 0 0 = 0
```

As discussed in Section 4.3, this subtle difference can under certain circumstances cause compilation failures upon automatic translation from HASKELL to C++. However, as far as the relative order of definitions is the concern, the real notoriety faced for such a translation is right here. Consider a translation from C++ to HASKELL which naïvely adds the two new *gcd* cases **after** the existing cases, and in particular, after the general case: No error/warning messages may be emitted in either language; both implementations are furthermore logically correct; yet, the subtle difference between languages entails observing different results. The reason will be arduous to spot and is likely to cater a full genre of implementation bugs. Automatic translation from C++ to HASKELL will be even trickier for the *correct* order of patterns needs to be figured out.

4.2 Representing Rational Numbers

For a compile-time representation of $\mathbb{Q}$, the nominator and denominator need to be stored as template parameters. Note that C++ template metaprograms are **pure** functional entities. As a result, the passed template arguments cannot be mutated, and the cancelled nominator and denominator have to be stored as nested values/types. We choose to represent those using types to avoid possible linkage failures:

```
template<long unsigned p, long unsigned q = 1, bool negative = false>
struct Rational
{
  BOOST_STATIC_ASSERT((q != 0));
  const static long unsigned gcd = GCD<p, q>::value;
  typedef mpl::integral_c<long unsigned, p / gcd> num_t;
  typedef mpl::integral_c<long unsigned, q / gcd> den_t;

  static string to_string()
  {
    return p?
    (
      (negative? "-": "") +
      lexical_cast<string>(num_t::value) +
      (den_t::value == 1?"": "/" + lexical_cast<string>(den_t::value))
    ):
    "0";
  }
};
```

In order to give equal range to the nominator and denominator, we use unsigned types for both. That entails storing the sign of a fraction in a separate Boolean template parameter (`negative`).[5] On the other hand, in HASKELL, data structures are represented using algebraic datatypes:

[5] Had we chosen the nominator to carry the sign of the fraction by making it signed, the nominator's maximum absolute value was enforced to be half that of the denominator.

```
1 data Fraction = Fraction Word Word Bool
2 Fraction _ 0 _ = undefined
```

The first important difference between the C++ version and the HASKELL one is right in how they deal with partially defined datatypes. In C++, one usually employs static assertion to outlaw undefined instantiations. (See the use of `BOOST_STATIC_ASSERT` at line 4 in the definition of `Rational`.) Such a mechanism is not present in HASKELL, so for example, line 2 in the definition of `Fraction` will not prevent its use with 0 as the denominator. HASKELL however does support partial definition of functions. Thus, one way to circumvent this problem is to encapsulate the outlawing in a pivotal function:

```
cancel (Fraction _ 0 _) = undefined
cancel (Fraction p q n) = Fraction (p ‘div‘ g) (q ‘div‘ g) n
  where g = gcd p q
```

Note that, in the C++ version, default values are provided for the denominator and sign, whereas that is not possible in HASKELL. Furthermore, because HASKELL does not support OOP, there is no way to store the cancelled nominators and denominators in the body of the type `Fraction`. Likewise, in HASKELL, one may go about string representation of a fraction as follows – again, not encapsulated in `Fraction`:

```
instance Show Fraction where
  show = show’ . cancel where
    show’ (Fraction 0 _ _) = "0"
    show’ (Fraction p q n) = (if n then "-" else "") ++
      show p ++ (if q == 1 then "" else "/" ++ show q)
```

The routine string representation is similar in C++ and HASKELL: A 0-nominator fraction is always written as “0” with no sign; when the denominator of a fraction is “1”, only the nominator is written; the sign is only written if the fraction is negative.

Due to the lack of local storage in HASKELL, the pattern of cancelling fractions before arithmetic operations will occur repeatedly. Obviously, this on-the-fly cancellation is unacceptable in C++ where efficiency is critical. Cancellation is an example of when, in the translation from HASKELL to C++, one would need to encapsulate the functionality in a (data) member for efficiency reasons. We anticipate that figuring out when similar encapsulations are needed is a highly non-trivial task for automatic translation, let alone the correct encapsulation.

In C++, the detachment of the string representation from the `Rational` class is considered inappropriate because that would be unreasonably scattered. Therefore, an automatic translation from HASKELL to C++ would need to be able to infer when to encapsulate such a function in the class. Likewise, because this encapsulation is not directly possible in HASKELL, one needs to detach similar member functions upon an automatic translation from C++ to HASKELL.

We finish this subsection by two basic functions which will be handy later:

```
num (Fraction p _ _) = p
den (Fraction _ q _) = q
```

4.3 Operations

C++ was never specifically designed to be a language for metaprogramming. A consequence is that metafunctions do not come with a built-in *return* mechanism. Hence, one needs to specify the deliverables using **named** nested types/values. We did already see the use of nested values for returning the value of a *gcd*. Here, we are going to use a type to represent the result of an operation on $\mathbb{Q}$:

```
template<...> struct Plus{typedef Rational<...> result;};
```

An automatic translation from HASKELL to C++ needs to be particularly careful on whether to map a given value to a nested value or a nested type.

Reducing Repetition Using Preprocessor Metaprogramming. Having to specify deliverables using nested entities makes dealing with expression combination very labouring. For example, if we choose to implement plus like

```
template<long unsigned p1, long unsigned q1, bool n1,
         long unsigned p2, long unsigned q2, bool n2> struct Plus {...};
```

there will be no way to perform operations like `Plus<Plus<...>, ...>`. The reason is that `Plus<...>` is not a `Rational` itself – its `result` nested type is. To circumvent this problem, one would provide a general case where both template parameters are arbitrary types (with particular nested types). One would then append that by the appropriate number of template specialisations.

In our case, for example, one would provide a general `template<typename T1, typename T2> struct Plus` in addition to specialisations for `Plus<T1, Rational<p2, q2, n2> >`, `Plus<Rational<p1, q1, n1>, T2>`, and `Plus<Rational<...>,Rational <...> >`. The trick is that, for all cases, when the template parameter is not a `Rational`, one **forward**s the operation to the respective `result` nested type. The real calculation happens only in the `Plus<Rational<...>, Rational<...> >` case. This repetitive *nested-entity forwarding* will occur for all the $\mathbb{Q}$ operations and C++ programmers often evade similar manual implementations using preprocessor metaprogramming:

```
#define OP_INIT_TEMPLATES(r, NestedType, OpName)  \
template<typename T1, typename T2>                \
struct OpName                                     \
{                                                 \
  typedef typename OpName<typename T1::NestedType,\
                        typename T2::NestedType>::NestedType NestedType;\
};                                                \
template<typename T1, long unsigned p2, long unsigned q2, bool n2>\
struct OpName<T1, Rational<p2, q2, n2> >          \
{                                                 \
  typedef typename OpName<typename T1::NestedType,\
                        Rational<p2, q2, n2> >::NestedType NestedType;\
};                                                \
template<long unsigned p1, long unsigned q1, bool n1, typename T2>\
struct OpName<Rational<p1, q1, n1>, T2>           \
{                                                 \
  typedef typename OpName<Rational<p1, q1, n1>,   \
                           typename T2::NestedType>::NestedType NestedType;\
};
#define RATIONAL_ARITHMETIC_OPS (Multiplies, (Plus, (Minus, BOOST_PP_NIL)))
#define RATIONAL_COMPARATIVE_OPS (Less, (EqualTo, BOOST_PP_NIL))
BOOST_PP_LIST_FOR_EACH(OP_INIT_TEMPLATES, result, RATIONAL_ARITHMETIC_OPS)
BOOST_PP_LIST_FOR_EACH(OP_INIT_TEMPLATES, type, RATIONAL_COMPARATIVE_OPS)
```

Here is a quick explanation on how that is particularly related to FP: Note first that the list of operation names (RATIONAL_ARITHMETIC_OPS and RATIONAL_COMPARATIVE_OPS) are constructed in exactly the same way as one would construct a list using the familiar *cons* operator in FP. Next, one needs to note that BOOST_PP_LIST_FOR_EACH is in fact the preprocessor equivalent of the famous *map* function in FP.

Given that HASKELL functions do already have built-in support for specifying deliverables, this entire code bloat is redundant when it comes to translation from C++MP to HASKELL. Figuring that out does not seem to be an easy task for automatic translation. The other translation direction is in fact not needed to generate the preprocessor portion because a machine can generate all classes themselves without getting bored. Yet, the translator needs to know that this code bloat is indeed needed for metafunctions enforce nested-entity forwarding.

In sizeable industrial projects where C++MP is needed, a careful mixture of template and preprocessor metaprogramming is often unavoidable. In such cases, deciding on how to deal with each part of the mixture is yet another non-trivial task for automatic translation. See Section 5 for more.

Comparatives The general idea to examine equality of two fractions is to examine the respective cancelled nominators and denominators whilst also taking the sign into account. Special cases can be handled quicker. Two fractions can obviously not be equal when they have different signs. However, our cancellation algorithm does not change the original sign of a Fraction. There comes a subtle consequence: Although the following two equations hold for every non-zero *q*:

```
Rational<0, q, false>::num_t::value == Rational<0, q, true>::num_t::value
Rational<0, q, false>::den_t::value == Rational<0, q, true>::den_t::value
```

the fractions would still have different signs. A pointwise equality test will therefore not quite work. The case for two 0 fractions with different signs needs special care. Let us examine the HASKELL implementation first:

```
instance Eq Fraction where
  (Fraction 0 _ _) == (Fraction 0 _ _) = True
  (Fraction _ _ n1) == (Fraction _ _ n2) | (n1 /= n2) = False
  f1 == f2 = (num f1' == num f2' && den f1' == den f2') where
    f1' = cancel f1
    f2' = cancel f2
```

Because HASKELL always chooses the first successful pattern match, no conflict happens between the first two cases. The story is different in C++ though for, in C++, there is no relative ordering between the in-scope specialisations. Hence, the following attempt will fail because neither specialisation is a better fit for equality between Rational<0, q, true> and Rational<0, q, false>. An ambiguity error message will be emitted for such a comparison attempt. It is noteworthy that an automatic translation from the HASKELL version to C++ is also most likely to produce something like:

```
template<long unsigned q1, bool n1, long unsigned q2, bool n2>
struct EqualTo<Rational<0, q1, n1>, Rational<0, q2, n2> >
{typedef mpl::true_ type;};

template
<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
> struct EqualTo<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{typedef mpl::false_ type;};
```

The solution is to merge the two specialisations into one. Expecting an automatic translation from HASKELL to C++ to manage this merging technique is not realistic. It would be even less expectable for the other direction of automatic translation to sort the correct HASKELL ordering out. The case when the two fractions have the same sign is routine and we will present all that together:

```
template
<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
> struct EqualTo<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{typedef typename mpl::bool_<p1 == 0 && p2 == 0> type;};

template
<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
> struct EqualTo<Rational<p1, q1, n>, Rational<p2, q2, n> >
{
  typedef typename Rational<p1, q1, n>::num_t num1_t;
  typedef typename Rational<p2, q2, n>::num_t num2_t;
  typedef typename Rational<p1, q1, n>::den_t den1_t;
  typedef typename Rational<p2, q2, n>::den_t den2_t;

  typedef typename mpl::bool_
  <
   num1_t::value == num2_t::value && den1_t::value == den2_t::value
  > type;
};
```

In HASKELL where Type Classes are available, our definition of `==` also implements `/=` automatically. Was our `Rational` class a runtime one, C++ Concepts could have likewise been used. Given that `Rational` is for compile-time use, we would need to define a new Concept to be the compile-time counterpart of EqualityComparable. We would like to remind that EqualityComparable defines the types and (runtime) functions each of its instantiations need to provide. In other words, EqualityComparable does not constrain the respective metafunctions of its instantiations. Were we about to go for the metaprogramming counterpart of the `Eq` Type Class of HASKELL, we needed to first define the Concept for types with metafunctions providing a `type` nested-type. Next, we had to define a Concept like MetaEqualityComparable, which states the names of the relevant equality metafunctions. We would need to implement `NotEqualTo` in terms of `EqualTo` in MetaEqualityComparable and state that `Rational` models it too. The definition of `EqualTo` for `Rational` would then suffice to get `NotEqualTo` automatically.

Alternatively, we can manually define operations in terms of each other without resorting to Concepts. Here, we only present how to do that for `NotEqualTo` in terms of `EqualTo`. Note that in the implementation of `Less`, similar merge techniques to the ones used for `EqualTo` are needed to avoid ambiguity between template specialisations.

```
template<typename T1, typename T2> struct NotEqualTo
{typedef typename mpl::not_<typename EqualTo<T1, T2>::type>::type type;};
```

A note on laziness seems suitable here: We could have chosen to implement the operations *lazily*, say using techniques presented in [2]. For example, we could have chosen `NotEqualTo` to be implemented as:

```
template<typename T1, typename T2> struct NotEqualTo
{struct apply{typedef typename mpl::not_<...>::type type;};};
```

The benefit of this technique is that the mere instantiation of `NotEqualTo<T1, T2>` will not trigger the computation. Instead, `NotEqualTo<T1, T2>` can be freely passed around with the computation only taking place the first time that `NotEqualTo<T1, T2>::apply::type` is queried. (Note the similarities with the (**bind**) in Figure 1.) Whether an automatic translation from HASKELL to C++ should by default choose the lazy metaprogramming or the strict one can be controversial. See Section 5 for more.

Arithmetics Arithmetic operations on types are typically expected to at least consist of the four basic operations. The metaprogramming techniques used for the implementation are mainly similar. So, we only present plus here in which the main FP technique used is mutual recursion with minus:

```
template
<
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
> struct Plus<Rational<p1, q1, n>, Rational<p2, q2, n> >
{
  typedef typename Rational<p1, q1, n>::num_t num1_t;
  typedef typename Rational<p2, q2, n>::num_t num2_t;
  typedef typename Rational<p1, q1, n>::den_t den1_t;
  typedef typename Rational<p2, q2, n>::den_t den2_t;

  typedef Rational
  <
   num1_t::value * den2_t::value + num2_t::value * den1_t::value,
   den1_t::value * den2_t::value,
   n
  > result;
};
template
<
```

```
 long unsigned p1, long unsigned q1,
 long unsigned p2, long unsigned q2, bool n
> struct Plus<Rational<p1, q1, n>, Rational<p2, q2, !n> >
{
  typedef typename mpl::if_c
  <
   n,
   typename Minus<Rational<p2, q2, !n>, Rational<p1, q1, !n> >::result,
   typename Minus<Rational<p1, q1, n>, Rational<p2, q2, n> >::result
  >::type result;
};
```

Here is a quick recap:

- When p_1/q_1 and p_2/q_2 both have a sign n, we first acquire the respective cancelled fractions through the nested types. Let us call them p'_1/q'_1 and p'_2/q'_2, respectively. (Note that these are already calculated at the instantiation time.) The result is $\frac{p'_1 \times q'_2 + p'_2 \times q'_1}{q'_1 \times q'_2}$ with sign n.
- Otherwise, the operation will transform to a subtraction. The resulting fraction has the same sign as that of the first fraction when that is negative. It takes the opposite sign otherwise. We would need to subtract $p_1/q_1$from p_2/q_2 in the latter case.

Perhaps the only syntactic difference between the C++ metaprogram and the HASKELL program is the use of guards in the HASKELL one in contrast to the use of inline operations. This is because in HASKELL in order to test whether the two fractions have the same sign or not, we cannot repeat the sign token in the pattern whereas this is fine in C++. So long as the guard conditions are simple, an automatic translation should not face much difficulty. Yet, complicated guards might need a separate treatment. Despite all that, the two codes are so similar that one could simply rewrite one with the other syntax to got to the other implementation.

```
plus (Fraction p1 q1 n1) (Fraction p2 q2 n2) | n1 == n2 =
  cancel (Fraction (num1 * den2 + num2 * den1) (den1 * den2) n1) where
    f1 = cancel (Fraction p1 q1 n1)
    f2 = cancel (Fraction p2 q2 n2)
    num1 = num f1
    num2 = num f2
    den1 = den f1
    den2 = den f2
plus (Fraction p1 q1 n1) (Fraction p2 q2 n2) | n1 /= n2 =
  cancel raw_result where
    raw_result = if n1
      then minus (Fraction p2 q2 n2) (Fraction p1 q1 n2)
      else minus (Fraction p1 q1 n1) (Fraction p2 q2 n1)
```

The subtle difference is that, for better efficiency in the HASKELL version, one would apply a final cancellation to `raw_result` when $n_1 \neq n_2$. However, this is not needed in the C++ version for the cancelled nominators and denominators will be automatically stored upon instantiation. In this very case, automatic translation might not be expected to be able to handle this subtle difference. In larger applications, however, real performance hits upon translation can source from similar subtle differences.

Samples Complicated arithmetic expressions can now be evaluated using our libraries.

```
typedef Rational<2, 3> r1t;
typedef Rational<1, 3, true> r2t;
typedef Rational<5, 3> r3t;
typedef Rational<10, 9, true> r4t;
typedef Rational<3, 5> r5t;
typedef LessEqual
<
 Plus<Multiplies<r4t, r5t>::result, Negate<r2t>::result>::result,
 Minus<Divides<Negate<r4t>::result, r1t>::result, r3t>::result
>::type result;
```

Note that all computations are compile-time. The corresponding runtime HASKELL is:

```
r1 = Fraction 2 3 False
r2 = Fraction 1 3 True
r3 = Fraction 5 3 False
r4 = Fraction 10 9 True
r5 = Fraction 3 5 False
result =
  (plus (multiplies r4 r5) (neg r2)) < (minus (divides (neg r4) r1) r3)
```

5 Conclusion

In this paper, we show how close C++MP is to FP – a fact known for a long time but never well presented. We implement a C++ compile-time abstract datatype for $\mathbb{Q}$ which closely corresponds to its HASKELL runtime counterpart. As also shown in this paper, despite the remarkable proximity in the techniques, the HASKELL syntax by far outperforms. Earlier research therefore suggests starting from the HASKELL programs as a design phase and moving to C++MP as the implementation. The idea is that working with the neat syntax of HASKELL is much easier than C++MP, especially for C++ templates are known to be notoriously unapproachable when it comes to error/warning messages. Yet, it is worth noting that software development typically entails several iterations between design and implementation before each release. The ability of going back and forth between the design (HASKELL programs in this case) and the implementation (C++ metaprograms in this case) here becomes vital. We argue therefore that any mechanical translation across the two languages needs to be bidirectional.

With this mindset, alongside the codes that we present in this paper, we consider translations both from HASKELL to C++ and vice versa. We study commonalities which pave the way for mechanical translations, as well as differences as the hindrances on the way. We show how pattern matching, tail recursion, and immutability is closely similar across HASKELL and C++MP. Figure 2 summarises the impediments caused by the differences enumerated in this paper. Along with the section visited in, it comments on the feasibility of overcoming each impediment. Here is a row-by-row discussion:

1. The general case should be moved to the top from HASKELL to C++, and to the bottom in the opposite direction of translation. The correct ordering for HASKELL should be provided by the user for the first time and should be retained for further back and forth translation so long as the user does not change it.

	C++	HASKELL	In	Sol
1	general case first, ordering of the rest irrelevant	first pattern match chosen	4.1	S
2	static assertion for partially defined ADTs	N/C	4.2	C
3	object encapsulation for compile-time ADTs	N/C	4.2	S
4	default values for template parameters	N/C	4.2	N/C
5	compile-time data members	N/C	4.2	C
6	N/C (named nested entities)	built-in support for specifying function deliverables	4.3	S
7	type representation for values vs real nested values	N/C	4.3	S
8	mixed preprocessor/template C++MP	N/C	4.3	S
9	no relative ordering between specialisations	first pattern match chosen	4.3	S

Sol = Solution, S = Semi-Automatically Possible, C = Circumventable, N/C = No Correspondent

Fig. 2. Mechanical Translation between HASKELL and C++MP

2. From C++ to HASKELL, one needs to resort to partially defined functions with pivotal role, if any. The translation needs to be instructed about the static assertion being artificially encapsulated in a function in the HASKELL equivalent.
3. From HASKELL to C++, the translation should be instructed about the functions to be packed together in an abstract datatype. In the other direction, each member function will be translated to a stand-alone function.
5. From C++ to HASKELL, functions which perform the stored calculation should be contrived to be used on-the-fly every time the compile-time data member is used in the C++ version. The translator needs to be instructed about this correspondence between the data members and functions for next translation iterations.
6. Use named nested entities in C++. The translator should be instructed not to translate these entities into HASKELL.
7. From HASKELL to C++, the user needs to manually advise the translator about each entity being translated into a nested value or a nested type. From C++ to HASKELL, every nested entity is translated into a HASKELL value.
8. If possible, from C++ to HASKELL, the user needs to manually instruct which parts of this mix to dismiss upon translation. From HASKELL to C++, the translator needs not to update the dismissed parts. Note that, in many cases, this dismissal might not be possible or very tricky to specify. See the supplementary notes below.
9. When upon the translation from HASKELL to C++ a compile error is emitted due this difference, use the merge technique presented in this paper or similar ones in C++. Usages of these techniques need to be marked for the translator to know not to touch the manually corrected translations until there is a change in the HASKELL version. Refer to guideline 1 for retaining the order of specialisations.

This paper makes it obvious that full mechanical translation between HASKELL programs and C++ metaprograms is not realistic. Nevertheless, Figure 2 and the above discussion do indeed suggest the approachability of semi-automatic translation. In order to incarnate that, one might be intrigued to try simulating features of one language in the other one. We would like to remind, however, that the whole point of resorting to HASKELL is to harness the spontaneity of C++MP's syntax for FP purposes. In other words, the syntax of C++MP is already exotic enough to make C++ programmers practice a variety of non-trivial indirections. Adding extra layers of indirection on top of that for the purpose of emulating HASKELL would defeat the purpose by entailing extra syntactic chaos. Likewise, because HASKELL is **designed to be uniparadigm**, emulating say OOP often produces very unnatural HASKELL codes.

F# is a strict FP language with selective laziness. Moreover, impediments 2 to 5 of Figure 2 will not be faced upon translation between C++MP and F#. This is because F# supports OOP too and one can employ: exceptions thrown in F# constructors for the second one; F#'s built-in object encapsulation for the third; abstract datatypes with multiple constructors for the forth; and, ordinary data members for F# abstract datatypes for the fifth impediment. Fully removing impediment 8 on mixed template/preprocessor metaprogramming can still be considerably demanding for F# too. Here is an instance of where in the industry one of the authors needed to use such a mixture: Consider a C++ struct `S` templatised by an integer and with a function call operator which, for every *n*, invokes a function `f` for the first *n* elements of an array `a`:

```
#define DUMMY_INDEXER(z, n, data) data[n]
#define CALL_WRAPPER_MACRO(z, n, unused)                        \
  template<> struct S<n>                                        \
  {                                                             \
    template<typename Array> double operator () (const Array& a)\
    {return f(BOOST_PP_ENUM(n, DUMMY_INDEXER, a));}             \
  };
```

Our final remark is that, although F# is a better match for facilitating C++MP by leveraging its FP nature, the best option is perhaps to design a new FP language with built-in correspondent FP features as that of C++MP. This is because: real-world C++ is often full of such complex mixtures; C++MP combines laziness and strictness in unusual ways that do not fully correspond to any available FP language; pattern matching in C++ uses the best match strategy as opposed to the common first match of FP; and, there is no point in trying to emulate the other language's features for better translation. A successful such language can then eventually become a part of the C++ tool chain.

References

1. *International Standard ISO/IEC 14882:1998(E): Programming Languages — C++* , (1998).
2. David Abrahams and Aleksey Gurtovoy, *C++ Template Metaprogramming: Concepts, Tools, and Techniques from Boost and Beyond (C++ in Depth Series)*, Addison-Wesley Professional, 2004.
3. Matthew H. Austern, *Generic Programming and the STL: Using and Extending the C++ Standard Template Library*, Addison-Wesley professional computing series, Addison-Wesley Longman Publ. Co., 1998.
4. Jeremy Gibbons, *Datatype-Generic Programming*, Spring School on Datatype-Generic Programming (Roland Backhouse, Jeremy Gibbons, Ralf Hinze, and Johan Jeuring, eds.), Lecture Notes in Computer Science, vol. 4719, Springer-Verlag, 2007.
5. Aleksey Gurtovoy and David Abrahams, *The Boost C++ Meta-Programming Library*, `www.boost.org/doc/libs/release/libs/mpl/doc/paper/mpl_paper.pdf`.

6. Joel de Guzman, Dan Marsden, and Tobias Schwinger, *Boost Fusion Library: Library for working with tuples, including various containers, algorithms, etc.*, `http://www.boost.org/doc/libs/release/libs/fusion/doc/html/index.html`.
7. Vesa Karvonen and Paul Mensonides, *The Boost Library, Preprocessor Subset for C/C++*, `http://www.boost.org/doc/libs/release/libs/preprocessor/doc/index.html`.
8. John Launchbury, *A natural semantics for lazy evaluation*, POPL '93: Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, ACM, 1993, pp. 144–154.
9. Bartosz Milewski, *Bartosz Milewski's Programming Cafe: Concurrency, Multicore, C++, Haskell*, `http://bartoszmilewski.wordpress.com/`.
10. Eugenio Moggi, *Notions of Computation and Monads*, Information and Compututation **93** (1991), no. 1, 55–92.
11. Eric Niebler, *Boost Proto Library: Expression template library and compiler construction toolkit for domain-specific embedded languages*, `http://www.boost.org/doc/libs/release/libs/proto/index.html`.
12. Paul Moore, *Boost Rational Library: A rational number class*, `http://www.boost.org/doc/libs/release/libs/rational/index.html`.
13. Rinus Plasmeijer and Marko van Eekelen, *Concurrent* CLEAN *language report (version 2.0)*, December 2001, `http://www.cs.kun.nl/~clean/contents/contents.html`.
14. David Sankel, *Algebraic Data Types Series*, `http://cpp-next.com/archive/2010/07/algebraic-data-types/`.
15. Seyed H. HAERI, *Observational Equivalence and a New Operational Semantics for Lazy Evaluation with Selective Strictness*, Proceedings of International Conference on Theoretical and Mathematical Foundations of Computer Science (TMFCS-10), 2010.
16. Ábel Sinkovics, *Functional Extensions to the Boost Metaprogram Library*, Proceedings of the Second Workshop on Generative Technologies (WGT) 2010, Electronic Notes in Theoretical Computer Science, vol. 264, Jul 2011, pp. 85–101.
17. ______, *Nested Lamda Expressions with Let Expressions in C++ Template Metaprorgams*, WGT'11 (Zoltán Porkoláb and Norbert Pataki, eds.), WGT Proceedings, vol. III, Zolix, 2011, pp. 63–76.
18. Ábel Sinkovics and Zoltán Porkoláb, *Expressing C++ Template Metaprograms as Lambda Expressions*, Trends in Functional Programming (Zoltán Horváth, Viktória Zsók, Peter Achten, and Pieter Koopman, eds.), Trends in Functional Programming, vol. 10, Intellect, UK/The University of Chicago Press, USA, June 2–4 2009, pp. 1–15.
19. Ádám Sipos, Zoltán Porkoláb, Norbert Pataki, and Viktória Zsók, *Meta<Fun>: Towards a Functional-Style Interface for C++ Template Metaprograms*, Tech. report, Eötvös Loránd University, Faculty of Informatics, Dept. of Programming Languages, Pázmány Péter sétány 1/C H-1117 Budapest, Hungary, 2007.
20. Erwin Unruh, *Prime number computation*, 1994, ANSI X3J16-94-0075/ISO WG21-462.
21. Marko van Eekelen and Maarten de Mol, *Mixed lazy/strict graph semantics*, Implementation and Application of Functional Languages, 16th International Workshop, IFL04 (Lüebeck, Germany) (C. Grelck and F. Huch, eds.), Technical Report 0408, Christian-Albrechts-Universität zu Kiel, September 2004, pp. 245–260.
22. Todd Veldhuizen, *Expression Templates*, C++ Report **7** (1995), no. 5, 26–31.
23. Vicente J. Botet Escribá, *Boost Ratio Library: Compile time rational arithmetic*, `http://www.boost.org/doc/libs/release/libs/ratio/index.html`.
24. Zoltán Porkoláb and Ábel Sinkovics, *C++ Template Metaprogramming with Embedded Haskell*, Proceedings of the 8th International Conference on Generative Programming and Component Engineering (GPCE 2009) (New York, NY, USA), ACM, 2009, pp. 99–108.